DiESeL

Stefano D'Incà Levis

Alla mia famiglia e alla mia

ragazza che mi sono sempre stati vicini,

sostenendomi e aiutandomi.

Indice

Capitolo 1

Introduzione

L'oggetto di questa tesi riguarda il refactoring e lo sviluppo di una libreria per il supporto generico di server distribuiti all'interno della rete PariPari. Tale libreria, già precedentemente sviluppata è chiamata DiESeL, acronimo per *"Distributed Extensive Server Layer"*.

La parte iniziale del lavoro si è concentrata sul refactoring di *DiESeL* e sulla risoluzione dei problemi di funzionamento presenti; terminata tale fase si è passati all'estensione delle funzionalità della libreria aggiungendo il multi-server. Infine si è sviluppata la versione che da la possibilità di qualificare i server e i nodi tramite le *"Feature"*, delle caratteristiche completamente arbitrarie quali per esempio banda o capacità di storage, che permettono di definire le specifiche che dovrà avere un server, e valutare quanto un nodo contribuirà a far raggiungere tali spespecifiche. Questa aggiunta permette di modificare le politiche di aggregazione dei nodi, andando a migliorare le prestazioni dei server distribuiti creati utilizzando DiESeL.

L'esposizione del lavoro è articolata come segue:

Capitolo 2 viene presentata la struttura del progetto "PariPari", con una breve descrizione delle funzionalità dei moduli che lo compongono.

Capitolo 3 viene illustrata in dettaglio la libreria DiESeL.

Capitolo 4 vengono illustrati gli obiettivi del lavoro.

Capitolo 5 vengono presentate le le scelte effettuate per raggingere gli obiettivi sopracitati.

Capitolo 6 viene proposta la realizzazione vera e propria del lavoro, analizzando anche alcune parti del codice.

Capitolo 7 viene dato spazio ad alcune riflessioni sul lavoro svolto ed ai possibili sviluppi futuri.

Capitolo 2

Il Progetto PariPari

Il progetto PariPari prevede la creazione di un'applicazione peer-to-peer multipiattaforma e multifunzionale anonima, offrendo servizi che variano da quelli caratteristici di internet, quali e-mail, DNS, web hosting, file hosting, chat IRC, a quelli delle reti P2P come il file sharing, Voip, etc. [1] Gli obiettivi principali che sono stati prefissi sono:

- garantire il funzionamento e la qualitá dei servizi sopracitati in una rete che esula dal paradigma client-server;
- sfruttare al massimo le potenzialitá offerte dal P2P per fornire nuovi servizi;
- rendere conveniente l'utilizzo di questa applicazione rispetto a quelle giá presenti sul mercato.

In questo capitolo sarà effettuata una panoramica del progetto e in particolare degli aspetti che caratterizzano maggiormente il nostro software, partendo dall'architettura di rete, passando per l'architettura di base del client espandibile e multiunfunzionale, fino a toccare il sistema di gestione dei crediti e dell'anonimato.

Il linguaggio di programmazione utilizzato da PariPari è Java. Questo offre una maggior sicurezza dell'applicazione ed è indipendente dalla piattaforma garantendo una portabilità elevata. Per cercare di gestire la collaborazione di un discreto numero di persone (ad ora circa una quarantina) si è puntato sull'introduzione dell'Extreme Programming e sulla standardizzazione del codice e della documentazione. Grazie a tali paradigmi infatti è possibile mantenere salda la coordinazione di tutto il gruppo e rendere più facile e vantaggiosa la manutenzione e l'ampliamento del programma. Tutti fattori necessari in un progetto open source dove ogni programmatore, anche esterno a PariPari, deve avere gli strumenti per sviluppare rapidamente i propri Plugin.

Riguardo l'utilizzo di Java un'altra caratteristica di rilevanza notevole è la possibilità di lanciare PariPari direttamente dal browser dell'utente, impiegando Java Web Start la quale:

- permette di scaricare ed eseguire le applicazioni java direttamente dal web;
- garantisce che venga sempre eseguita l'ultima versione dell'applicazione;
- elimina le procedure di installazione e aggiornamento che spesso risultano ostiche.

Java tuttavia presenta anche dei risvolti negativi, uno su tutti la lentezza. Infatti, essendo un linguaggio interpretato, le istruzioni Java prima di essere eseguite dalla macchina vengono interpretate dalla JVM (Java Virtual Machine); per eseguire ogni istruzione l'elaboratore eseguirà un numero di istruzioni macchina che è più del doppio delle istruzioni che eseguirebbe se la stessa istruzione fosse stata scritta in C. Questo non è un problema da poco in una rete in cui si vuole garantire l'anonimato

attraverso l'uso della crittografia che richiede un elevato costo computazionale e comporta quindi un calo delle prestazioni. Fortunatamente le dimensioni degli stream di byte da cifrare sono basse e questo rende meno visibile all'utente finale l'effettivo rallentamento nell'esecuzione. [2]

2.1 Architettura di rete

Gli svantaggi che presenterebbe una architettura centralizzata sarebbero molteplici per una applicazione che intende fornire i nostri serizi: i server rappresenterebbero dei punti critici, la cui presenza sarebbe fondamentale per mantenere la struttura della rete, e nel caso di indisponibilità la rete collasserebbe; inoltre una architettura centrale è afflitta da problemi di scalabilità, infatti all'aumentare del numero di utenti si rende necessario aggiungere dei nuovi server per evitare un sovraccarico; infine la presenza di server rende la rete più vulnerabile agli attacchi esterni del tipo denial of service.[1] Per ovviare a questi problemi, l'architettura adottata da *PariPari* è priva di server. Infatti in una architettura P2P i problemi di scalabilità sono di più facile gestione, essendo tutti i nodi considerati alla pari. Altresì gli utenti possono rimanere connessi per periodi brevissimi (o arbitrariamente lunghi) senza creare problemi. La struttura generale della rete di *PariPari* si basa su una variante dell'algoritmo Kademlia, molto diffuso nei software peer-to-peer (come per esempio Emule).[6] A differenza dell'algoritmo originale, in *PariPari* è stata apportata una modifica con l'obiettivo di ridurre il tempo di ricerca delle risorse all'interno della rete. Di seguito si fornisce una breve spiegazione di come

[1] Denial Of Service (DOS): un attacco che mira a rendere inutilizzabile un servizio tramite un elevato numero di richieste alla macchina o alle macchine che lo ospitano, fino alla saturazione delle risorse disponibili

funziona l'algoritmo di Petar Maymounkov e David Mazières e successivametne si illustreranno le modifiche che sono state apportate a Kademlia in *PariPari*.

Kademlia è un protocollo di rete peer-to-peer che indicizza allo stesso modo host (chiamati ID) e risorse (chiamate Hash), e permette di ricercare entrambi nella rete in maniera completamente decentralizzata. Il concetto fondamentale su cui si basa Kademlia è quello della distanza, che permette di determinare la "prossimità" tra due nodi; la vicinanza o lontananza è calcolata utilizzando la metrica XOR. Kademlia tratta i nodi come se si trovassero su un albero binario, le cui posizioni sono determinate dal più corto prefisso del suo ID. In figura 2.1 si può vedere la posizione di un nodo con prefisso unico 0011. Kademlia assicura che ciascun nodo conosca almeno un nodo in ciascun sottoalbero della rete (in figura 2.1 sono cerchiati in grigio) se questo sottoalbero contiene nodi. Grazie a questa garanzia, ciascun nodo può localizzare nella rete un qualsiasi altro nodo tramite l'ID.

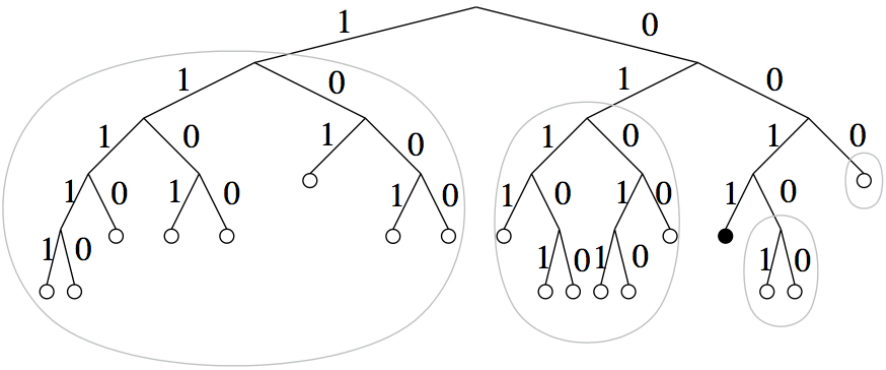

Figura 2.1: Albero binario di una rete Kademlia. In nero è evidenziata la posizione di un nodo 0011... nell'albero. Cerchiati in grigio ci sono i sottolaberi in cui il nodo 0011... deve avere un contatto.

ID e Hash hanno lo stesso formato, quindi è possibile determinare la distanza tra un nodo e una risorsa esattamente come si calcola la distanza tra

due nodi. Di conseguenza per un nodo è sempre possibile determinare quale sia, tra due chiavi, quella più vicina a lui o ad un qualsiasi altro nodo. Nel momento in cui un nodo vuole condividere una risorsa ne calcola l'Hash, ricerca nella rete gli host con ID più prossimo, in metrica XOR, all'Hash della risorsa, ed infine richiede a tali host di memorizzare le informazioni per raggiungerla. La ricerca di una risorsa avviene mandando contemporaneamente la richiesta a 3 nodi fra quelli da esso conosciuti con ID più vicino all'Hash desiderato e il nodo richiedente riceve al massimo 20 riferimenti ai nodi conosciuti con ID più vicino all'hash. Fra questi riferimenti, che possono essere al massimo 60, vengono scelti i 3 più vicini all'Hash cercato e il richiedente a questo punto inoltra la richiesta ai nodi a cui questi 3 puntano. Il procedimento continua fino a che viene raggiunto il riferimento al nodo ospitante la risorsa.

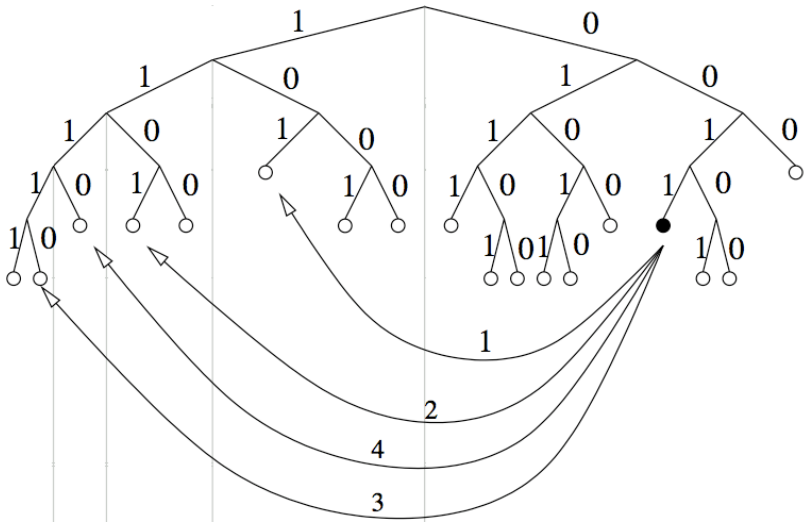

Figura 2.2: Passi dell'algoritmo in Kademlia dal nodo 0011... al nodo 1110... mediante successive query.

In figura 2.2 è schematizzata la ricerca in Kademlia dal nodo 0011... al nodo

1110... mediante richieste successive al nodo con ID più vicino nei sottoalberi via via più piccoli. Ad ogni salto la dimensione del sottoalbero di ricerca si dimezza. Essendo i sottoalberi di ricerca via via più piccoli si ha la certezza che l'algoritmo converga. La ricerca può essere vista come un cammino all'interno di un albero, e quindi la sua complessità è dell'ordine del numero di salti necessari per passare da una foglia ad un altra, ovvero proporzionale all'latezza dell'albero che è *O(log n)*.

Per ottenere prestazioni migliori si è pensato di introdurre una nuova funzionalità, che ridurrà il numero delle comunicazioni. Il primo nodo, infatti, che chiameremo origine, oltre a comportarsi come descritto nell'articolo di Petar Maymounkov e David Mazières[6], sceglie, tra i nodi che deve contattare, un prediletto. Questo "prediletto" inoltrerà direttamente la richiesta di ricerca a uno dei nodi che sta per trasmettere come risposta all'origine. Questo avviene iterativamente ad ogni salto della ricerca: il prediletto di prima generazione sceglie un prediletto tra i suoi nodi, mentre trasmette la propria risposta direttamente all'origine. In questo modo, nel caso la catena di prediletti non si interrompa, si arriva quasi a dimezzare il tempo di ricerca .

Col metodo standard, per permettere all'origine di conoscere l'indirizzo del nodo cercato sarebbero necessarie un numero di comunicazioni uguale a

$$comunicazioni = (distanza - 1) * 2$$

Invece, con il nuovo sistema, potrebbero bastare

$$comunicazioni = distanza - 1$$

Assumendo che tutte le comunicazioni abbiano uguale durata, si avrebbe un risparmio del 50% in termini di tempo. In caso di insuccesso, d'altra parte, le

prestazioni non verrebbero assolutamente variate, dato che la ricerca continuerebbe in parallelo seguendo il metodo standard.

2.2 Struttura del client PariPari

Per accedere ad una risorse di *PariPari* non è obbligatorio utilizzare la nostra applicazione. *PariPari* appare all'esterno come un'unica macchina virtuale, la cui realizzazione interna è trasparente ai non appartenenti alla rete. L'approccio adottato per la stesura del codice é quello a Plugin. Il Plugin é un programma non autonomo che interagisce con un altro programma per ampliarne le funzioni. Grazie a questa strategia architetturale, ogni modifica apportata al singolo Plugin sará trasparente al resto delle altre sezioni. Inolte una modularitá cosí elevata garantisce un punto di forza notevole anche nel caso in cui le modifiche apportate siano sostanziali.

Nell'architettura di *PariPari* si possono differenziare due diversi tipi di Plugin: quelli della cosidetta cerchia interna (Core, Crediti, Connettività, Storage e DHT) e quelli esterni (IRC, DNS, Voip, Emule, Torrent, etc...). I primi si occupano di gestire la coordinazione dei Plugin e di fornire le risorse basilari per il loro funzionamento. I secondi offrono invece servizi specifici e si basano sull'utilizzo dei Plugin della cerchia interna. Il termine Plugin può quindi essere utilizzato per indicare una parte di *PariPari* dedicata allo svolgimento di una determinata funzione. I Plugin sono tra loro indipendenti ed il Core si occupa di gestirli e di assegnare loro le risorse richieste. Non vi è quindi comunicazione diretta fra Plugin ma essi devono necessariamente passare per il core, che si occupa inoltre di impedire l'inondazione di richieste da parte di

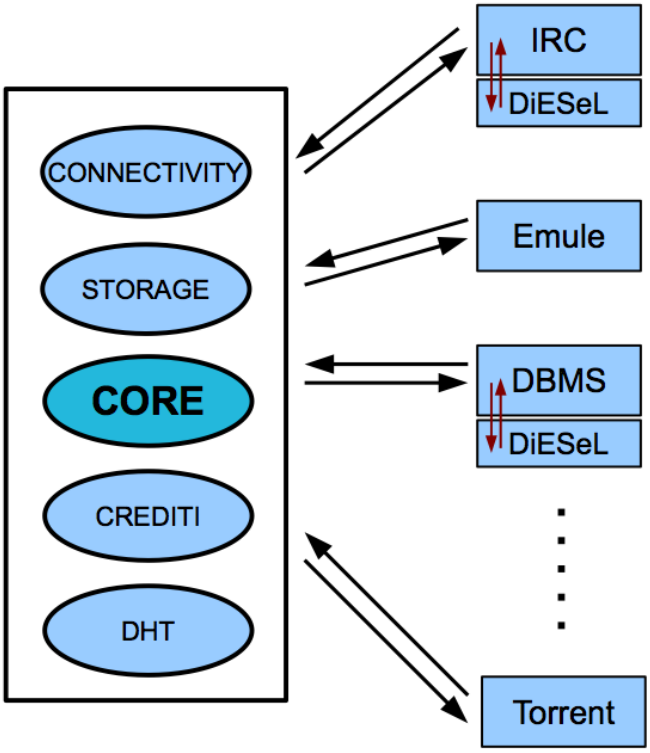

Figura 2.3: Schema della suddivisione dei Plugin in *PariPari*. I plugin IRC e DBMS utilizzano la libreria *DiESeL* per distribuire i loro server sulla rete.

eventuali Plugin maligni e di evitare che questi prendano il possesso di risorse destinate ad altri. In particolare qualora un Plugin intenda richiedere una risorsa ad un altro dovrà inviare una richiesta standard al Core, specificando il Plugin di destinazione e la risorsa richiesta. Il Core esaminerà la richiesta ed eventualmente la passerà al destinatario, che risponderà fornendo la risorsa. È stato adottato questo approccio per permettere ad ogni sviluppatore di usare gli altri Plugin come delle black-box, senza cioè conoscerne i dettagli realizzativi.

2.3 Sistema dei Crediti

La gestione dei crediti in una rete P2P è uno dei punti cruciali poichè regola i rapporti di collaborazione dei nodi e garantisce quindi la sua esistenza. In molte reti P2P è presente un meccanismo di assegnazione dei crediti che ha lo scopo di incentivare gli utenti a rimanervi il più a lungo e di condividere il maggior numero di risorse possibili. In reti come eDonkey2000 il sistema è piuttosto rudimentale e funziona in maniera diretta: se Alice scarica un file da Bob e in quel momento Bob non ha bisogno di nulla da Alice, Alice memorizzerà il suo debito nei confronti di Bob e non appena Bob richiederà qualche risorsa ad Alice, quest'ultima la esaudirà tempestivamente.

Il sistema dei crediti implementato in PariPari si ispira al libero mercato. Si assuma che Bob, Alice, Charlie siano tre utenti della rete. Alice e Bob spesso concludono affari tra loro così come Alice e Charlie, mentre Bob e Charlie non hanno mai avuto nessun contatto diretto. Si immagini ora che Charlie sia interessato ad acquisire una risorsa di Bob. Purtroppo, Charlie non è in credito con Bob e non ha modo per aggiudicarsi tale risorsa. Alice, però, è fortemente indebitata con Charlie. Charlie procederà quindi ad acquisire la risorsa grazie alla mediazione di Alice, che risulterà essere in debito con Charlie e in credito con Bob. Alice si troverà ad avere crediti verso diversi altri utenti che le verranno richiesti con frequenze e quantità sempre diverse. Il suo scopo sarà quello di riuscire a compiere queste mediazioni, guadagnando comunque sulla transazione, non rimanendo mai sprovvista di quei crediti che potrebbero servirle in futuro. L'utente ha inoltre la possibilità di distribuire fra i vari Plugin diverse percentuali dei crediti accumulati. In tal modo, macchine

particolarmente ricche di una specifica risorsa (ad esempio, molta memoria disponibile), possono guadagnare i crediti necessari per usufruire di tutti gli altri servizi.

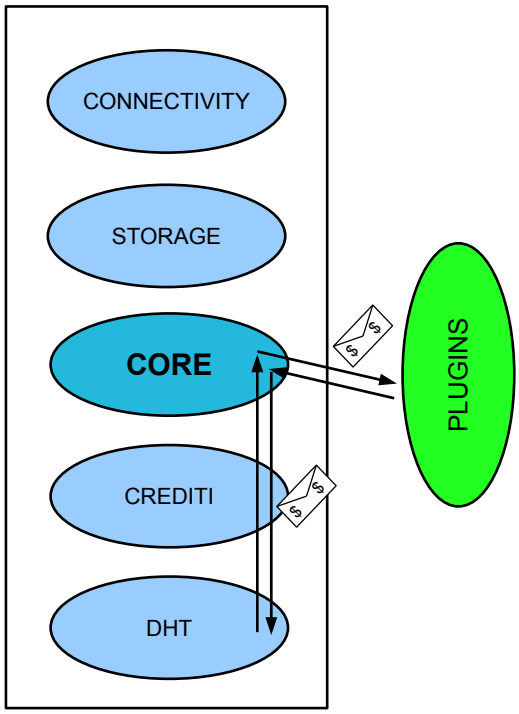

Figura 2.4: Esempio di utilizzo dei crediti. É rappresentata schematicamente la richiesta di una risorsa di *DHT* da parte dei Plugins.

Nel caso in cui un nuovo nodo entri nella rete, viene stabilito che non possa indebitarsi, senza prima aver sprecato una certa quantità delle sue risorse. Il nuovo utente che volesse acquisire una risorsa della rete, non avendo nulla con cui scambiarla, e neppure altri crediti per procedere a una mediazione, sprecherebbe, per esempio, la sua banda, come pegno della sua buona volontà, per comprare la risorsa desiderata. È immediato realizzare che la quantità sprecata non deve essere eccessivamente grande per non precludere l'accesso alla rete, nè troppo piccola per

fornire un efficace deterrente ai possibili nodi sanguisuga che potrebbero sferrare un attacco Sibilla alla rete.[2]

Nella progettazione dei singoli Plugin bisogna dunque tener conto della limitata disponibilità di risorse, e cercare di agire col massimo della parsimonia. Nel caso in cui un Plugin non abbia sufficienti crediti per eseguire una richiesta, questa viene direttamente respinta dal core, fino all'eventuale sospensione del Plugin stesso. Questo accorgimento impedisce ad eventuali Plugin maligni di inondare il core con continue richieste non soddisfabili al fine di interrompere il funzionamento degli altri Plugin.

[2]Lo scopo dell'attacco Sibilla è quello di guadagnare credibilità in un sistema basato su reputazioni mediante la votazione da parte di un gran numero di nodi creati appositamente dal nodo malevolo.

Capitolo 3

DiESeL

Figura 3.1: Logo della libreria DiESeL.

DiESeL, acronimo per Distributed Extensive Server Layer, è una libreria che permette di gestire la distribuzione di applicazioni di tipo server sulla rete *PariPari*

indipendentemente dalle funzionalità svolte. La libreria andrà a creare un substrato al di sotto del livello applicativo del server garantendone la distribuzione su più nodi della rete. *DiESeL* è stato sviluppato in modo tale da assicurare la scalabilità, trasparenza e stabilità del server distribuito.

I punti cruciali di *DiESeL* sono diversi, e nei successivi paragrafi verranno analizzate le scelte fatte per affrontarli in maniera efficace.

3.1 Interfacciare DiESeL con Server Layer

Il primo problema che si pone è quello di rendere la libreria completamente indipendente dal server che la utilizzerà; perciò è stata definita un'interfaccia che permette al ServerLayer di usufruire di *DiESeL* in modo trasparente. L'interfaccia in questione è stata chiamata *IServerLayer* che dovrà essere implementata dalle classi che intendano usare la libreria ricreando i metodi base per la comunicazione con il *DistributorLayer* e con gli altri nodi sulla rete. Lo scambio di informazioni tra i due livelli è reso possibile attraverso una coda, accessibile in mutua esclusione, dove l'applicazione del ServerLayer può depositare i messaggi da inviare agli altri nodi del server distribuito; tale invio è reso possibile dal broadcast messo a disposizione da *DiESeL* il quale incapsula il messaggio in un Object generico e lo inoltra a tutti. Al momento, in attesa di sviluppi da parte del modulo di connectivity, il broadcast è eseguito in maniera "brutale": il nodo di *DiESeL* che riceve il messaggio lo invia direttamente a tutti gli altri nodi partecipanti al server come si può vedere in figura 3.2.

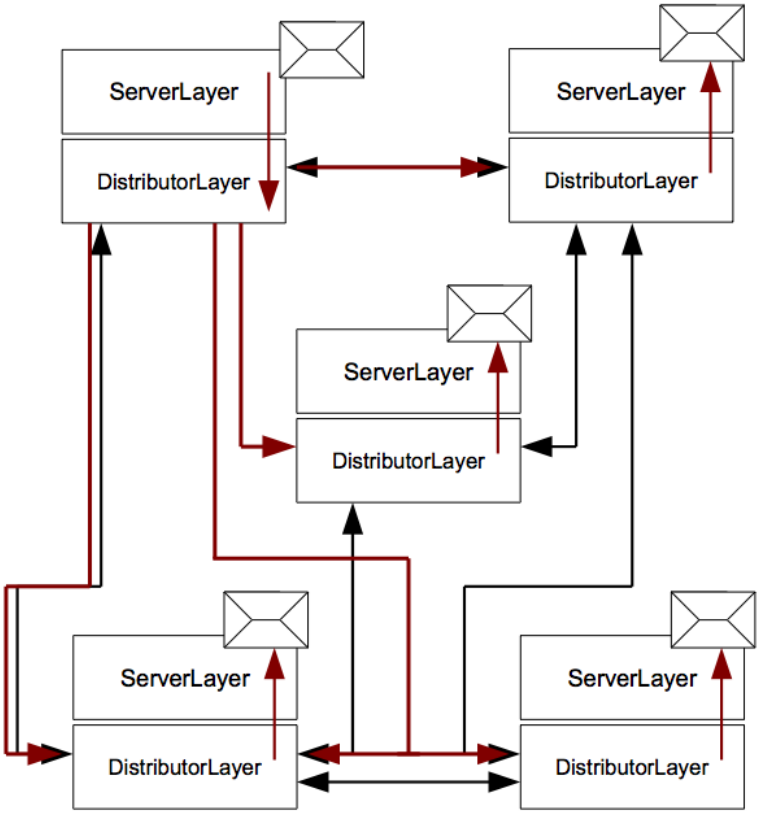

Figura 3.2: Broadcast dei messaggi ricevuti dal *ServerLayer*. Si può notare che allo stato attuale il nodo che riceve il messaggio lo invia direttamente a tutti i partecipanti al server in maniera diretta.

3.2 Keep Alive e Outage Detection

Un altro problema particolarmente critico è quello del Keep-Alive, ovvero del mantenimento e del monitoraggio delle connessioni dei nodi facenti parte alla rete. Se non si prestano adeguate attenzioni si potrebbe causare un eccessivo overhead dovuto alle comunicazioni, comportando così problemi alla scalabilità. Il sistema di Keep-Alive deve essere in grado di riconoscere le disconnessioni (volontarie o meno) dei nodi e comunicarle agli altri nodi, prendendo le contromisure necessarie per sopperire alla dipartita dei nodi. In molte applicazioni si è notato come tali sistemi

generassero un overhead sulla rete inaccettabile (ad esempio nelle prime versioni di Gnutella era pari a circa il 50% del traffico totale generato [7]), soprattutto con l'aumento del numero di connessioni tra i nodi. In *DiESeL* si è cercato un compromesso tra il minimo numero di link tra i nodi, per garantire la connessione dell'intero server, e il minimo traffico generato dal sistema di Keep Alive, ottenendo comunque buone prestazioni sulla velocità di riconoscimento di un outage e sulla conseguente riparazione della sottorete. L'algortitmo utilizzato è il Cooperative Keep Alive (CKA).[8]

Negli algoritmi di Keep Alive Standard (SKA) solitamente due nodi connessi tra loro si scambiano, a periodi di K secondi, dei messaggi di Ping cui seguiranno delle Replies da parte del destinatario verso il mittente. Ogni nodo X avrà dunque n collegamenti bidirezionali con altrettanti nodi che definiscono la cosiddetta neighbourhood di X. In una rete di N nodi di grado medio d(dove con grado si intende il numero di connessioni per nodo) ci sarannoNd collegamenti bidirezionali, generando un overhead di messaggi ogni K secondi. Ciò comporterebbe con un numero elevato di connessioni medie per nodo d, un grosso incremento di traffico addizionale. Lo scopo del CKA è quello di permettere di aumentare d senza introdurre eccessivo overhead sulla rete. L'idea di base è quella di permettere che due nodi Y e Z, entrambi connessi a un terzo nodo X, possano collaborare per riconoscere un eventuale outage di X mantenendo la conoscenza dei link di X stesso. Più precisamente ogni nodo avrà una lista contenente i nodi a cui è connesso; avrà inoltre un thread che ogni K secondi contatterà il nodo in testa a tale lista, inviandogli una richiesta di Keep Alive, riprovando per K volte in caso di mancata ricezione di una risposta. In queste ultime sarà specificato anche il momento in cui dovrà essere nuovamente contattato. Qualora X non invii alcuna

Reply, il mittente Y riconoscerà l'outage del nodo X e instaurerà il meccanismo di flooding per avvisare gli altri nodi. L'algoritmo di Outage Detection deve tenere conto della possibilità che un nodo rilevi erroneamente un outage a causa di ritardi nella rete, e deve cercare, per quanto possibile, di minimizzare il traffico causato dalle comunicazioni del flooding. Nell'algoritmo CKA queste operazioni prendono il nome di *Sequential Flooding*. Quando un nodo Y rileva l'outage di un nodo X, informerà tutti i vicini di X a lui noti con un opportuno messaggio. I vicini, una volta ricevuto il messaggio, cercheranno a loro volta di contattare X per assicurarsi che esso sia caduto realmente e in tal caso risponderanno ad Y comunicando la propria lista di vicini di X noti. Y potrà dunque aggiornare la propria lista dei vicini di X in modo sequenziale e informare correttamente tutti i nodi interessati, limitando le inevitabili ridondanze generate nel caso in cui ogni nodo informasse i vicini a lui noti direttamente.

3.3 Struttura della sottorete del server distribuito

Fondamentale per la rete del server distribuito è la robustezza alle continue connessioni e disconnessioni dalla rete[1]. *DiESeL* quindi deve creare le connessioni tra i vari nodi della rete in modo tale che una eventuale caduta non causi un *net-split*. Per farlo, il numero delle connessioni tra i nodi deve essere sufficiente, ma non eccessivo da comportare uno spreco di banda, e tali connessioni non devono essere casuali, ma invece gestite dal modulo stesso al fine di aumentare la connessione del grafo della rete.

[1] Churn rate

Dalla teoria dei grafi casuali è noto che, per essere connesso con alta probabilità, ogni nodo di un grafo di taglia N deve avere grado superiore a $log(n)$, anche se in generale questo è vero sopratutto per N molto grande. Nel nostro caso quindi si è scelta come soglia minima di numero di connessioni per nodo $\lceil log(N) \rceil$ cercando di rafforzare tale decisione assegnando i link tra i nodi in maniera non casuale. L'idea è quella di connettere ogni nuovo nodo ad un numero logaritmico (rispetto al totale) di nodi scegliendo di volta in volta quello con grado minore e quindi con maggior probabilità di essere isolato. Nel caso si debba scegliere tra più nodi con uguale grado, la scelta iniziale sarà il nodo cui spetta la scelta delle connessioni; successivamente le scelte verranno effettuate in modo casuale cercando di evitare nodi vicini[2] ai nodi già scelti. In questa maniera si mantiene il grado dei vari nodi praticamente costante (con uno scarto massimo di poche unità tra il nodo con più link e quello con meno). Inoltre, avendo tutti i nodi egualmente connessi e assegnando i vari link secondo il criterio sopra descritto, si mantiene un livello di clustering molto basso, ovvero il grafo risulterà come un unico gruppo uniforme di nodi e non diviso in più sottogruppi, riducendo così il rischio di net-split.

Lo svantaggio di utilizzare questo sistema è che ogni nodo dovrà mantenere la tabella con il vicinato suo e di tutti i partecipanti al server, comportando così un maggiore utilizzo della memoria locale e un traffico aggiuntivo sulla rete causato dalle comunicazioni degli aggiornamenti dei vicinati.

[2]Per vicinanza dei nodi si intende il numero di link tra un nodo e l'altro all'interno della sottorete del server

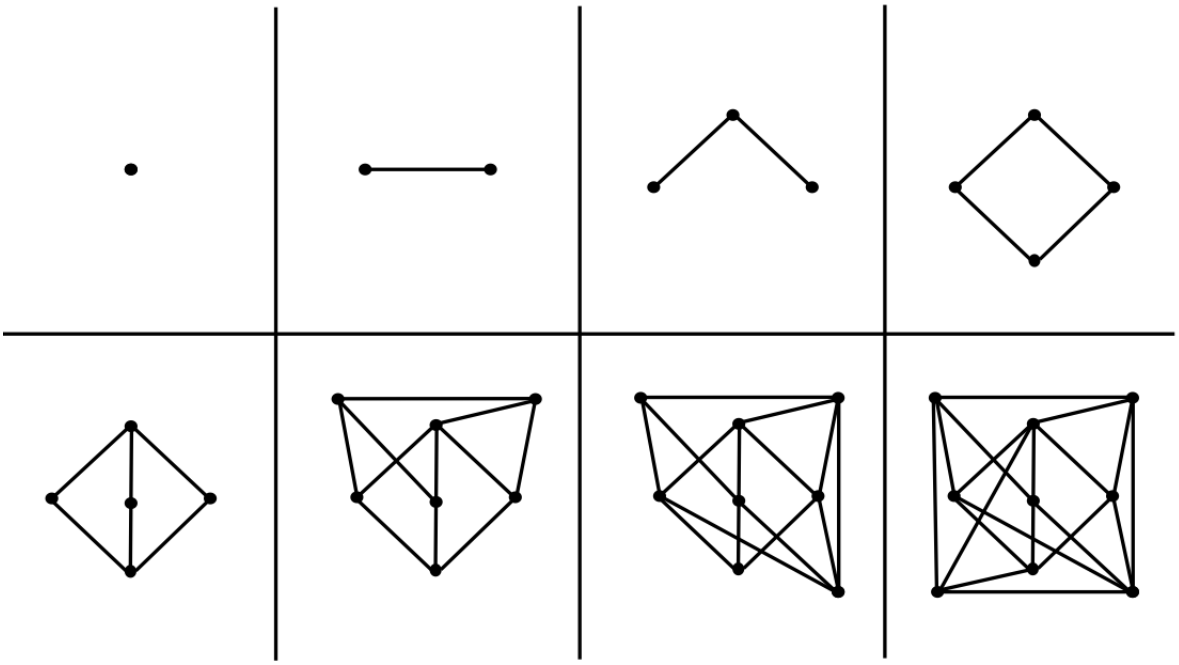

Figura 3.3: Composizione di un grafo con 8 nodi.

3.4 Gestione connessioni esterne

La caduta di un nodo del server, evento piuttosto probabile in una rete peer-to-peer, deve essere il più trasparente possibile agli eventuali client connessi in quel momento. Perciò la soluzione deve essere efficace.

L'idea di base è quella di nascondere l'outage di un nodo cercando di reindirizzare la connessione dei client su altri nodi possibilmente sui più stabili. Tra i possibili metodi ne sono stati presi in considerazione tre, molto noti, che sono:

Switching Consiste nell'incanalare i messaggi ricevuti su un certo nodo verso altri nodi ed è stata scartata a priori in quanto evidentemente non adatta. In questo modo infatti verrebbero a crearsi dei colli di bottiglia nei nodi che effettuano molte redirezioni e, con l'outage di uno di questi, verrebbero comunque perse le connessioni per cui esso fungeva da switch.

TCP Handoff Consiste nel manipolare a basso livello i pacchetti modificandone l'indirizzo IP (effettuando quindi una sorta di IP spoofing) al fine di spostare la connessione verso un indirizzo prestabilito. Questo meccanismo risulta completamente trasparente al lato client ma presenta lo svantaggio di dover andare a lavorare molto a basso livello (cosa non sempre semplice in Java).

DNS-Badsed Consiste nell'andare a gestire le connessioni grazie all'interazione con un server DNS. L'idea è basata sul fatto che il modulo *DiESeL* possa essere in grado di valutare la situazione del server distribuito e la bontà dei nodi che lo compongono, in modo tale da poterli ordinare in una lista di indirizzi. Tale lista verrà mantenuta aggiornata e comunicata al server DNS che la assocerà all'URL corrispondente. In questo modo, un client esterno che voglia appoggiarsi ad un server distribuito attraverso la propria URL verrà indirizzato dal DNS verso il nodo attualmente considerato più affidabile.

La soluzione scelta è stata quella basata sul DNS, andando a collaborale con il gruppo DNS di PariPari, che gestisce appunto lo sviluppo di un server DNS distribuito. Punto cruciale per l'utilizzo di questa politica di reindirizzamento dei client è la valutazione dei nodi. Tale "classifica" dovrà essere effettuata basandosi su una metrica che vada a determinare la probabilità del nodo di rimanere attivo per lungo tempo e la capacità di gestire ulteriori connessioni basandosi sul rapporto tra banda disponibile e numero di connessioni attuali. Attualmente tale metrica non è ancora stata definita e sarà oggetto degli sviluppi futuri.

Un ulteriore distinzione si deve poi fare tra i diversi tipi di client. I nodi di *PariPari* infatti potranno ricevere richieste di connessione sia da parte di client esterni sia da parte di client di *PariPari* relativi a una data applicazione. In

quest'ultimo caso, sarà possibile effettuare una riconnessione trasparente, a meno che il nodo del server non cada per cause accidentali. Nel caso di chiusura volontaria dell'applicazione da parte di un utente, si potrebbe infatti inserire un meccanismo che avvisi i client a lui connessi di puntare ad un altro nodo, senza che questo sia notato in alcun modo dall'utente.

3.5 Struttura generale di DiESeL

Il modulo *DiESeL* è composto da una classe principale che gestisce al suo interno una serie di thread. Ognuno di questi avrà un determinato compito che nel complesso contribuirà a rendere effettive le scelte implementative precedentemente descritte. Nella figura 3.4 si può notare la composizione del modulo. Ogni riquadro rappresenta una classe, le frecce indicano le relazioni logiche tra le classi stesse e corrispondono ai vari flussi di messaggi al proprio interno. Sono anche visibili le raffigurazioni delle code che verranno sfruttate in mutua esclusione dalle varie classi per depositare o ricevere i diversi messaggi su cui è basato il funzionamento di tutta la libreria.

La classe principale del modulo è stata denominata *Distributore* e in essa vengono gestiti i vari thread e le comunicazioni tra di essi. In particolare questa classe gestisce lo stato del *DistributorLayer* e le eventuali operazioni per la ricerca e l'attivazione di un nodo. Un nodo può essere attivato dal server layer in modalità "passiva" o in modalità "attiva" . Nel primo caso i nodi rimangono in attesa di venire contattati per partecipare ad un server, e nel momento in cui ne facessero parte, si limiterebbero ad eseguire i ping descritti dall'algoritmo CKA, ed eventualmente inoltrare agli altri nodi i messaggi passati dal

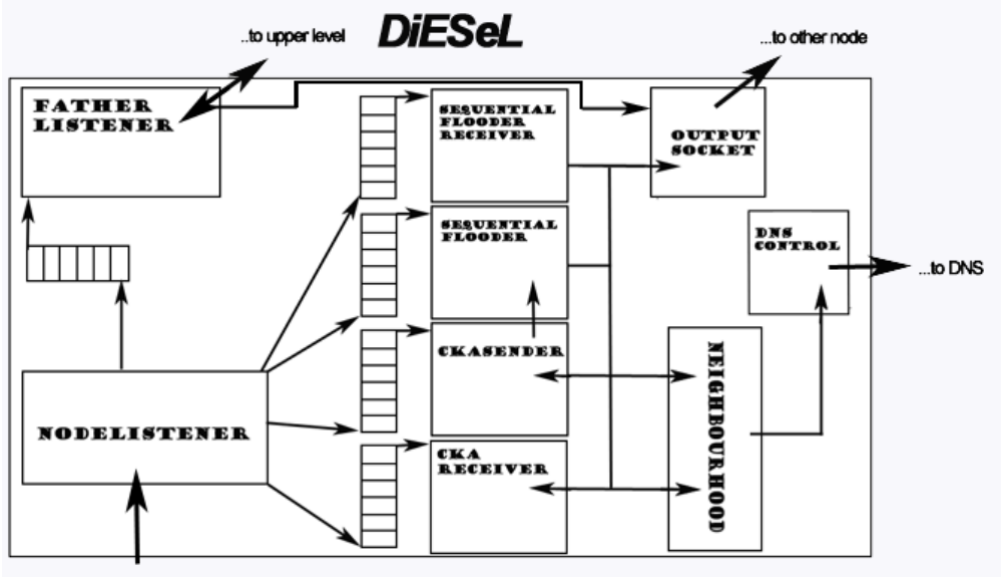

Figura 3.4: Schema di interazione delle classi del modulo *DiESeL*.

ServerLayer. Nel secondo caso invece, il nodo farà partire il server distribuito, e ne sarà il responsabile, gestendo l'aggregazione di altri nodi, se presenti, ad intervalli regolari. Nel momento in cui il nodo "comandante" dovesse disconnettersi per qualche motivo, il nodo che ne rileverà l'outage si autoeleggerà "attivo", ereditando i doveri del suo predecessore. Anche il nodo "attivo" esegue i ping necessari all'algoritmo CKA ed inoltra se necessario i messaggi passati dal *ServerLayer*.

Le altre classi presenti in figura 3.4 sono le seguenti:

FatherListener è il thread responsabile delle comunicazioni con il *ServerLayer*. Si limita a inoltrare i messaggi che vengono depositati dal livello superiore su una coda di Object a tutti i nodi del server.

CKASender e CKAReciever sono i therad che realizzano l'algortitmo CKA.

SequentialFlooder e SequentialFlooderReciever sono i due thread che eseguono l'algoritmo di Sequential Flooding per l'outage detection come descritto nel paragrafo 3.2.

NodeListener è il thread che una volta ricevuto il PingMessage lo ricompone dal flusso di byte e lo identifica; una volta individuato il tipo di messaggio il thread lo deposita nella rispettiva coda in modo che gli altri thread possano leggerlo. Se riconosce una reply ad un ping deposita il messaggio nella coda condivisa con il CKASender, se è un messaggio di outage nella coda condivisa con il SequantialFlooderReceiver, se è una risposta ad un messaggio di outage nella coda condivisa con il SequentialFlooder, se è un messaggio per il ServerLayer nella coda condivisa con esso. Un caso particolare è quello dei messaggi di ping. Normalmente essi saranno depositati nella coda condivisa con il CKAReceiver ma, in alcuni casi, in base al valore assunto dal campo piggyBack del messaggio verranno effettuate azioni differenti e il messaggio sarà scartato. Questo avviene con le richieste e gli invii di messaggi di backup (BACKUP), con le informazioni inerenti al vicinato di alcuni nodi (NEIGHBOURINFO), con il comando di chiusura del nodo (KILL!), con l'informazione riguardante il nuovo nodo in modalità di comando (COMMAND) e con le richieste e rispettive risposte per verificare l'esistenza di un nodo durante il Flooding Sequenziale (CONFIRM e CONFIRMREPLY).

OutputSocket è il thread che si occupa di prendere i messaggi destinati ad un altro nodo, convertirli in un flusso di byte e inviarli.

Neighborhood è la tabella in cui vengono mantenute le informazioni della rete: ad ogni nodo sono associati i suoi "vicini". La struttura dati utilizzata è la

DiESeLHashTable, ovvero una tabella *Hash* implementata dagli sviluppatori di *PariPari*.

DNSController è un thread che periodicamente dovrebbe controllare lo stato dei nodi sulla rete e formulare una lista ordinata di indirizzi per comunicarla al modulo DNS di *PariPari*. Attualmente, a causa dello sviluppo ancora limitato di alcuni moduli di *PariPari*,questo thread non viene utilizzato, e non è stata formulata alcuna metrica per la discriminazione dei nodi.

Come precisato in precedenza, solo il nodo "attivo" A ha la possibilità di aggregare altri nodi al server. Per farlo va ad interagire con il Plugin DHT di *PariPari*, richiedendogli tutti i nodi pubblicati sulla DHT che hanno dato la disponibilità ad unirsi al server. Ogni Plugin che intenda utilizzare *DiESeL* per distribuire un server ha la possibilità di crearne uno solo, ma come verrà illustrato nei prossimi capitoli, parte di questa tesi è stata dedicata a permettere la creazione di un numero arbitrario di server per ogni Plugin. Una volta che DHT ha restituito i candidati, il nodo A ne sceglierà uno (nodo B) e proverà a contattarlo. Nel primo messaggio saranno inseriti il riferimento temporale del server, fondamentale per sincronizzare i ping necessari all'algoritmo di CKA, l'identificativo del nodo comandante[3] e la lista dei nodi a cui il nuovo partecipante dovrà connettersi (come spiegato nel paragrafo 3.3). In figura 3.5 si può vedere l'handshake tra i due nodi.

Ricevuto questo messaggio il nodo B richiederà il backup al nodo a cui si connetterà. Quest'ultimo richiederà a tale richiesta richiedendo al ServerLayer

[3]Ogni nodo è identificato in maniera univoca da una struttura dati, l'*IDiESeLNode*, che contiene l'InetAddress e le due porte necessarie ai soket per comunicare. E' possibile che 2 nodi nella rete abbiano lo stesso indirizzo ip, quindi per poterli differenziare si utilizzano anche le porte.

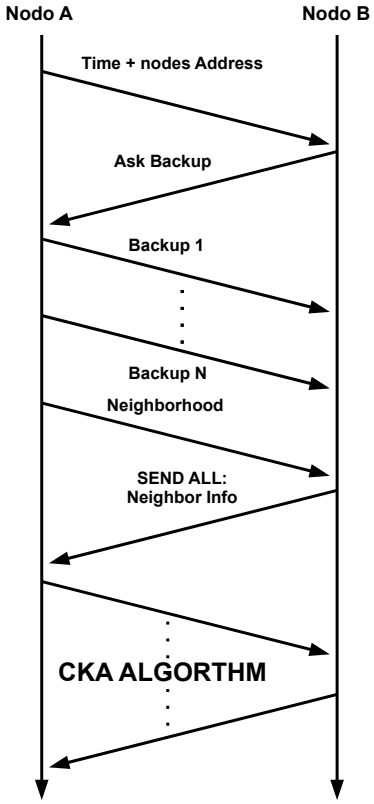

Figura 3.5: Handshake di un nuovo nodo del server.

un backup delle strutture dati necessarie ad ogni nodo per ottenere la consistenza dei dati iniziale, e inviando gli oggetti ottenuti in risposta al nodo B attraverso dei messaggi di ping in cui il campo piggyBack specifica il particolare contenuto del messaggio. Oltre al backup, viene inviata anche la Neighborhood. A questo punto il nodo B aggiornerà la sua tabella dei vicini, ed invierà a tutti i nodi della rete le parti modificate. A questo punto il nodo B invierà al suo nuovo "vicino" il primo ping necessario all'algortimo CKA. A questo il nodo che lo riceve risponderà con un ping in cui specificherà al nodo B l'istante in cui dovrà pingarlo nuovamente e contemporaneamente inseri-

rà nella coda contenente i ping che deve effettuare il ping destinato al nodo B. Questa serie di PING/REPLY continuerà sino a quando uno dei due nodi non si disconnetterà, e l'outage farà partire la procedura di Sequential Flooding con cui verranno informati tutti i nodi della sua dipartita (come spiegato nel paragrafo 3.2).

Capitolo 4

Obiettivi del lavoro

La parte iniziale dell'attività di tesi si è concentrata sul refactoring del classi che presentavano difetti di funzionamento, andando ad eliminare tutti i problemi della libreria e rendendo così finalmente operativo *DiESeL* (paragrafo 4.1). Successivamente, è stato aggiunto il multiserver. In precedenza i servizi che utilizzavano *DiESeL* potevano creare un unico server a cui partecipavano tutti i nodi del dato servizio (si chiarirà meglio nel paragrafo 4.2); ora invece è stata data la possibilità ad un qualunque servizio (ad esempio *IRC* o *DBMS*, ...) di creare un numero arbitrario di server ciascuno identificabile dal nome. I nodi che vogliono partecipare ad un server, possono scegliere a quale unirsi. Infine, sono state aumentate le potenzialità della libreria inserendo le *Features*, caratteristiche completamente arbitrarie[1], che vanno a modificare le politiche di aggregazione dei nodi al server. In particolare l'inizializzatore andrà a specificare le caratteristiche che dovrà avere il server, quali per esempio la banda, il numero di nodi che dovranno comporlo, o qualsiasi cosa possa essergli utile. A questo punto tutti i partecipanti dovranno

[1] L'unico vincolo che si pone è che siano numerabili.

specificare la loro disponibilità nel fornire ciascuna caratteristica, e l'aggregazione continuerà sino a quando le specifiche del server non saranno soddisfatte.

4.1 Refactoring

Parte fondamentale per poter svolgere il lavoro è rendere operativo *DiESeL*. Inizialmente la libreria funzionava esclusivamente con un massimo di tre nodi connessi; nel caso se ne fosse unito un altro, i nodi non riuscivano più a comunicare in maniera corretta, e nel caso di caduta del comandante, il server non riusciva a riadattarsi, causando un net-split. Il risultato inevitabile era la composizione di due sottoreti rispettivamente da uno e da due nodi. Inoltre si riscontravano dei problemi nella tabella che gestiva i vicini, e all'aumentare di questi, si avevano delle perdite di informazioni. La fase di debug non è stata semplice, a causa soprattutto della natura distribuita del server, che introduce problemi aggiuntivi. Infatti, a causa dei tempi necessari per l'inizializzazione (spesso dilatati a causa di problemi di interazione con altri moduli di *PariPari*) e per l'aggiunta dei nodi al server, ad ogni singola prova servivano svariati minuti.

I problemi riscontrati erano molteplici e di seguito sono riportati i principali:

- Come sopra riportato, si presentava un problema nella gestione dei vicini da parte della Neighborhood, causato da un funzionamento anomalo (o da un utilizzo sbagliato) della HashTable di Java. Si è tentato di comprendere la causa di questo malfunzionamento ed in fine il problema è stato risolto semplicemente implementando una *DiESeLHashTable*.

- Essendo un server distribuito, e poiché l'algoritmo di CKA necessita una

sincronizzazione nello scambio dei ping, è fondamentale avere una forma di temporizzazione per i nodi della rete. Infatti molti dei problemi erano causati da una scorretta gestione dei tempi nei vari host, e all'aumentare del numero di nodi il server non riusciva più ad avere un tempo comune. Inoltre al momento del ripristino in seguito alla caduta del comandante, il tempo del server non si aggiornava, comportando grossi problemi alla sincronizzazione e causando l'inevitabile net-split.

- La desincronizzazione dei nodi del server, unita ad un problema nel comunicare da parte del CKAReciever l'istante in cui venire contattato, comportavano, da parte di alcuni nodi, il mancato invio dei ping necessari all'algoritmo di Keep Alive. Spesso questa situazione provocava una mancata gestione della scomparsa dei nodi, generando una situazione instabile del server.

- Infine, l'operatività di *DiESeL* è stata limitata dall'interazione con altri moduli di *PariPari*; in particolare si riscontravano gravi problemi con il modulo *DHT*, fondamentale per la ricerca di altri nodi da aggiungere al server. Tali problemi si sono risolti in seguito al refactoring del modulo da parte del team di *DHT*.

Una volta risolti tutti i problemi, si è resa finalmente operativa la libreria ed è stato possibile implementare le modifiche di cui si parlerà nei prossimi paragrafi e capitoli.

4.2 Multiserver

Fino ad ora, nel momento in cui un Plugin di *PariPari*, utilizzando *DiESeL*, creava un server, tutti i nuovi nodi dovevano dare la disponibilità a connettersi a quell'unico server, e dopo un periodo di attesa, vi sarebbero stati aggregati. Com'è intuibile, l'utilità di questo modo di procedere risulta limitata in quanto potrebbe esserci la necessità di creare più server differenti che offrano lo stesso servizio (per esempio creare diversi server Web). Inoltre, aggregare tutti i nodi ad un unico server sarebbe uno spreco nel caso in cui questo non venisse sfruttato a sufficienza.

Si è pensato quindi di introdurre la possibilità di creare server distinti. In particolare, nel momento in cui il *ServerLayer* attiva un nodo, come già precisato, può crearlo in due diverse modalità: "comandante", o "attiva", oppure in modalità "passiva". Nel caso in cui il nodo sia passivo, questo dovrà ricercare se sono già presenti dei server che forniscono il suo stesso servizio (ovvero nel caso il *ServerLayer* sia stato implementato dal Plugin *IRC* il nodo ricercherà se esistono dei server *IRC*), e in tal caso sceglie a quale unirsi. A questo punto, il nodo rimarrà in attesa della chiamata da parte del nodo comandante del server a cui si è unito. Se invece il nodo è attivo, avrà la possibilità di creare un nuovo server specificandone il nome. Nel caso si presentassero delle omonimie si gestirà il problema aggiungendo al nome del nuovo server una stringa di caratteri casuali.

Come si vedrà nel paragrafo successivo, utilizzando le *Features* si potrebbe incorrere nell'indesiderata eventualità che dei nodi rimangano in attesa di venire contattati per un tempo troppo lungo. Per risolvere questo problema

si darà la possibilità al *ServerLayer* di creare dei nodi attivi con una particolare join policy: aggregare i nodi passivi in attesa a prescindere dal nome del server a cui hanno deciso di unirsi. Questo non comporta grossi problemi di sicurezza, in quanto aggregando nodi a intervalli regolari, non si corre il rischio che un server, con questa politica di unione, si impossessi di tutti i nodi.

4.3 *Features*

I server sono caratterizzati da una serie di specifiche, e a seconda dell'utilizzo che se ne vuole fare, queste caratteristiche variano: ad esempio un server casalingo non dovrà avere le stesse prestazioni di un server di una banca. Anche su *DiESeL* risulta molto utile poter specificare tali caratteristiche, denominate *Features*. Il primo problema che si incontra è quello della loro definizione: non è possibile sapere a priori quali saranno, in quanto, a seconda dell'utilizzo che si farà del server le caratteristiche saranno differenti. Si è deciso quindi di permettere al *ServerLayer* di dichiarare in maniera completamente arbitraria non solo la quantità, ma anche il tipo delle *Features* del server. L'unica limitazione che si pone alla scelta del tipo è che sia numerabile, e nel paragrafo 5.2 verrà spiegata tale scelta progettuale.

Come già detto la definizione delle specifiche sarà a carico del *ServerLayer*, il quale dovrà comunicarle a *DiESeL* al momento della creazione del server. Risulta ovvio che tutti i nodi partecipanti al server dovranno specificare la loro disponibilità per ciascuna *Feature*. Anche questo compito è lasciato al *ServerLayer*, il quale dovrà premurarsi di controllare la veridicità delle dichiarazioni fatte da ciascuno.

A questo punto sarà possibile distinguere i nodi in base alle *Features*, e così facendo si potrà modificare la politica di aggregazione in base alla disponibilità di ciascuno nodo di "donare" risorse al server, scegliendo ogni volta quello più adatto.

Questo modo di procedere porterà ad un migliore utilizzo delle risorse disponibili: innanzitutto non verranno uniti nodi senza limite, ma solo fino a quando la somma delle *Features* dei nodi partecipanti al server pareggeranno quelle richieste; in secondo luogo, potendo distinguere la "bontà" di un nodo, si andranno ad aggregare principalmente elementi che apporteranno un beneficio maggiore in termini di quantità di *Features*.

Capitolo 5

Idee per raggiungere gli obiettivi

In questo capitolo verranno presentate le idee per raggiungere gli obiettivi descritti nel capitolo precedente. Viene tralasciata la fase di debug, che pur essendo stata una parte sostanziosa del lavoro di tesi, non è ne di facile stesura per sua natura, ne molto interessante dal punto di vista accademico.

5.1 Multiserver

Per realizzare il multiserver, è necessario innanzitutto che i nodi siano distinguibili. Il comandante ricerca i nodi da aggiungere al server mediante un interrogazione al modulo *DHT*, specificando il servizio a cui devono appartenere. Per esempio, nel caso del Plugin *WebServer*, il nome del servizio sarà: *paripari.WebServer.DiESeL.Powered*. La *DHT* di *PariPari* permette di effettuare ricerche sui dati memorizzati, utilizzando più chiavi; quindi se al momento della pubblicazione del nodo sulla Distributed Hash Table, oltre al nome del servizio, si memorizza anche il nome del server, sarà possibile distinguere i nodi in base al

server a cui partecipano.

Ogni nuovo nodo che si connette seguirà quindi il seguente procedimento:

- Interrogazione al modulo *DHT* per determinare i server attivi dello stesso servizio.

- Nel caso il nodo sia attivo, verrà richiesto all'utente di inserire il nome del server desiderato.[1] Nel caso il nome sia già utilizzato, automaticamente si dovrà inserire una stringa casuale di caratteri per differenziarlo. Invece nel caso in cui il nodo sia passivo e non vi siano server pubblicati, per evitare una attesa troppo prolungata, automaticamente lo si promuoverà ad attivo comportandosi come sopra descritto. Nel caso vi siano server pubblicati si chiederà all'utente di scegliere a quale unirsi.

- Scelto il nome del server, lo si dovrà pubblicare nella Distributed Hash Table, insieme all'identificativo del nodo (Indirizzo IP e coppia di porte necessarie per le comunicazioni dei soket di *DiESeL*[2]) e al nome del servizio.

- Il nodo passivo rimarrà in attesa di venire contattato. Il comandante invece, ad intervalli regolari (ogni minuto), andrà a cercare nella *DHT* se vi sono memorizzati dei nodi che hanno dato la disponibilità ad unirsi al proprio server; in caso affermativo, ne contatterà uno (la politica di aggregazione verrà spiegata nel prossimo paragrafo e in maniera più approfondita nel paragrafo 6.2.3).

[1] Ci sarà la possiblità di definire il nome anche via codice, senza coinvolgere l'utente.
[2] L'identificativo dei nodi nella libreria è il *DiESeLNode*, composto come detto da IP e dalle 2 porte.

Nel caso in cui il server non sia creato da un utente, ma venga definito dallo sviluppatore del Plugin, si dovrà dare la possibilità di definirne il nome via codice. Risulta quindi necessario distinguere tali server e per farlo si dovrà creare una classe che ne descriva la tipologia.

L'interazione con *DHT* è parte centrale dello sviluppo del multiserver. Al fine quindi di facilitarne l'implementazione, e di ridurre la ripetizione del codice, si dovrà riprogettare alcuni metodi che comunicano con tale modulo di *PariPari*, ed in particolare quelli relativi alla ricerca delle risorse. Nel capitolo successivo verranno forniti i dettagli.

5.2 Features

L'introduzione delle *Features* porterà grandi vantaggi, dando al possibilità di differenziare sia i server che i singoli nodi. Come detto nel capitolo precedente non è conveniente, ne possibile, andare a definire quali saranno le caratteristiche. Per questo motivo, si è scelto di renderle completamente arbitrarie, sia nella forma che nella sostanza, ovvero è possibile deciderne sia il nome che il tipo di dato. In realtà, dopo attente analisi è stato valutato opportuno limitare il tipo di dati utilizzabili ai soli valori numerabili interi o in virgola mobile. Questa decisione è figlia della convinzione che le caratteristiche che contraddistingueranno un server, saranno necessariamente quantificabili e quindi esprimibili mediante dei numeri. Si è convinti che tale restrizione non ridurrà la potenza delle *Features*, e contribuirà a diminuirne la complessità di implementazione.

Sarà compito del *ServerLayer* richiedere le *Features* all'utente, o definirle via codice, in modo tale da mantenere l'indipendenza della libreria dai Plugin. Data la

natura estremamente variabile delle *Features*, si è pensato che il metodo migliore per farle comunicare dal *ServerLayer* a *DiESeL* è quello di scriverle in un file *.xml*, letto dalla libreria. Si è scelto il linguaggio *XML* per la sua duttilità e facilità.

I Plugin dovranno andare a definire per il nodo attivo due file distinti: quello in cui si troveranno le caratteristiche desiderate del server, e quello in cui saranno definite le specifiche del nodo. Per ciascun nodo passivo invece, dovranno creare solo un file dove saranno definire le sue *Features*. Ogni nodo quindi avrà le proprie caratteristiche e queste dovranno essere esattamente dello stesso tipo di quelle definite per il server. Nel caso contrario si dovrà sollevare un eccezione. In figura 6.2 si può vedere un esempio di uno dei due file. Durante l'inizializzazione del nodo, sia attivo che passivo, *DiESeL* memorizzerà le *Features* trovate nel file *.xml* e le pubblicherà nella *DHT*, insieme al suo identificativo (la stringa rappresentante il proprio *DiESeLNode*) e ad il nome del server a cui partecipa. Questo passaggio è fondamentale per la realizzazione delle politica di aggregazione: in seguito alla ricerca nella Distributed Hash Table dei nodi da unire al serve, il comandante avrà una lista di candidati; grazie alle *Features* potrà scegliere quello che in quel momento risulta il migliore (si veda il paragrafo 6.2 relativo ai dettagli implementativi).

Si noti che il comandante, responsabile delle annessioni, conoscerà tutte le *Features* di ciascun nodo (nel momento della scelta infatti le leggerà dalla *DHT*), e ne dovrà mantenere traccia per conoscere lo stato del server. Inizialmente si è pensato che bastasse conoscere solo la quantità complessiva, ma ci si è presto accorti che sarebbe stato un errore. Infatti nel momento in cui uno o più nodi si disconnettessero in maniera non volontaria, il dato delle *Features* complessive perderebbe di significato in quanto non sarebbe possibile risalirne alla quantità rimanente, se non andando ad interrogare tutti gli elementi rimanenti, comportando

un traffico aggiuntivo insostenibile data la natura della rete. Si è quindi optato per mantenere le caratteristiche di ciascun nodo nella Neighborhood. Per non appesantire le comunicazioni durante la fase di handsake (figura 3.5), la tabella dei vicini viene inoltrata "droppata" delle *Features*.

Come già evidenziato, sarà necessario che il nodo attivo mantenga traccia del quantitativo totale delle *Features* del server, sommando le caratteristiche dei nodi, nel caso di nuove connessioni o sottraendo nel caso di disconnessioni.

A questo puno si pone un altro problema: la caduta accidentale del nodo attivo. In questa situazione un nuovo nodo verrà eletto comandante, ma gli mancheranno tutte le informazioni necessarie per determinare lo stato del server, non potendo così realizzare una politica di aggregazione efficace. Dovrà quindi contattare direttamente tutti i nodi richiedendo le *Features*. Come è intuibile, questa procedura comporterà del traffico aggiuntivo, risultando un collo di bottiglia rallentando la ripresa del server. Un altra strategia possibile è quella di richiedere le caratteristiche interrogando la *DHT*, ma anche in questo modo non vengono risolti i problemi di traffico che causano il collo di bottiglia. Una strategia più intelligente è quella di distribuire le *Features* a tutti i nodi. Per evitare di sprecare banda, dovuta all'intestazione dei pacchetti si è deciso di utilizzare il piggyback degli inevitabili ping dell'algoritmo di Keep Alive. Mantenendo contenuta la taglia del pacchetto, non si andranno ad appesantire le comunicazioni e si potrà diffondere la conoscenza di tutte le caratteristiche. Così facendo, il nodo che eredita il comando del server, avrà già almeno una parte delle *Features* dei nodi (se non addirittura tutte) e le restanti le richiederà direttamente.

La strategia che verrà utilizzata per la distribuzione nella sottorete delle *Features*

si ispira alla teoria degli algoritmi epidemici.[3] A seconda della taglia del piggyback il nodo A inserirà nel ping le *Features* di uno o più nodi (se necessario anche solo parte). Tale nodo avrà una coda per ogni su vicino, in cui saranno presenti le *Features* che dovranno venirgli comunicate. Queste code saranno ordinate in maniera casuale, e nel momento in cui A riceve delle nuove caratteristiche andrà ad inserirle prima nella Neighborood ed in seguito nelle code, in posizione aleatoria. Al momento di inviare un ping al nodo B, A estrarrà dalla rispettiva coda le *Features* da comunicargli inserendole nel piggyback. Così facendo si avrà, a discapito di un piccolo appesantimento dei ping, l'eliminazione (o quanto meno una forte riduzione a seconda della conoscenza complessiva raggiunta dalla rete) del traffico aggiuntivo determinato dalla caduta del nodo attivo.

La politica di aggregare nodi sino a quando non siano raggiunte le specifiche richieste del server, può comportare un problema di busy waiting. Infatti un nodo che si mette a disposizione di un server che non necessita di altre risorse, rimarrà in attesa sino alla caduta di qualche partecipante. Questo può comportare una attesa breve, vista la natura delle reti peer-to-peer, ma anche arbitrariamente lunga. Per ovviare a questa situazione il *ServerLayer* avrà la possibilità di inizializzare dei server con una join poicy particolare (specificandola nella classe che lo descriverà): aggregare i nodi di cui necessita senza considerare il server a cui hanno deciso di unirsi. L'aggregazione temporizzata ogni minuto garantisce che i server con questa join policy non si approprino di tutti i nodi della rete.

[3]Gli algoritmi epidemici sono così chiamati in quanto tentano di disseminare le informazioni nella rete "infettando" il più alto numero possibile di nodi attraverso meccanismi di diffusione aleatoria.

Capitolo 6

Realizzazione

In questo capitolo sarà descritta nel dettaglio l'implementazione delle modifiche apportate a *DiESeL*. In particolare si andrà a presentare la struttura necessaria a supportare il Multiserver e successivamente le *Features*.

6.1 Multiserver

La prima parte dell'implementazione del multiserver consiste nel definire le possibili tipologie del server; questo viene fatto mediante l'utilizzo dell'"enumeratore" *ServerLayerDescriptionEnum* e la scelta verrà fatta in fase di implementazione da parte del *ServerLayer*. I casi possibili sono i seguenti:

NO_NAME_ONE_SERVER Esisterà un unico server del servizio a cui verranno aggregati tutti i nodi.

HAS_NAME_USERDEFINED Ci saranno più server, ciascuno con il nome deciso dall'utente che lo crea.

HAS_NAME_CODEDEFINED Ogni server avrà il nome deifinito dal codice.

HAS_NAME_NOT_DEFINED Ci saranno più server, ognuno con un nome generato casualmente.

Questo "enumeratore" è contenuto nella classe *ServerLayerDescription*, che conterrà quindi le informazioni del server, tra cui il nome.

Risulta fondamentale per la realizzazione del multiserver l'interazione con il modulo *DHT*. Infatti, per riuscire a trovarsi, due nodi devono interrogare la *DistributedHashTable* ricevendo le coordinate per iniziare la comunicazione. Fino ad ora veniva creato un unico server, e quindi l'unica discriminate sui nodi da contattare era il nome del servizio a cui appartenevano. Ora invece, è necessario pubblicare nella *DHT* oltre al servizio, anche il nome del server. Di seguito è riportata la porzione di codice che si occupa di memorizzare nella DHT queste informazioni:

```
[...]
Class<?> requestForAPI = DHTAPI.class;
IFeatureValue[] storeValue = new FeatureValue[1];
storeValue[0] = new FeatureValue("time", 30000);
IRequest req = new Request(requestForAPI, new
    ConstructorParameters(storeValue, DHTAPI.Primitive.
    STORE_PRIMITIVE, service, keys, notes, new byte[12]));
IReply reply = null;
try {
    reply = myGrandFather.askTheCore(req);
 }
[...]
```

Come si può notare, i dati che verranno memorizzati sono *service*, *keys* e *notes*.

Service É il nome del servizio, ovvero il nome del Plugin che utilizza la libreria concatenato alla stringa *DiESeL.Powered*.

Keys É la lista delle chiavi che verranno salvate e sono l'identificativo del nodo, ovvero l'IP più le due porte, e il nome del server.

Notes É la lista di informazioni che saranno utili alla libreria, ovvero le due porte del nodo necessarie per la comunicazione dei socket, il nome del server e le *Features*.

Nel caso in cui il nodo sia attivo e l'enumeratore sia "HAS_NAME_USERDEFINED", prima di procedere con il salvataggio nella *DHT* è necessario richiedere il nome del server all'utente; questo avviene mediante l'inserimento del nome in una finestra a pop-up, controllando che non vi sano casi di omonimia. Negli altri casi invece ("HAS_NAME_CODEDEFINED", ecc.) il nome o è passato dal codice, o non è necessario. Nel caso di un nodo passivo invece, si cercheranno i nomi di tutti i server esistenti del servizio e si richiederà all'utente di scegliere a quale unirsi; per facilitare la procedura verrà creata una finestra a pop-up con un menù a tendina, contenente la lista. L'utente dovrà quindi semplicemente "cliccare" sul nome desiderato.

I metodi che eseguono la ricerca sono i seguenti:

interrogationToDHT Questo metodo riceve come parametri la lista delle chiavi da ricercare nella DHT, e la modalità di ricerca (intersezzione o unione dei risultati), e restituisce la lista dei nodi responsabili delle risorse ricercate. La lista delle chiavi varia a seconda di cosa si sta cercando:

- nel caso si cerchino nodi da aggiungere al server, le chiavi saranno *nomePlugin.DiESeL.Powered* e *nomeServer*;

- nel caso si stiano cercando i server esistenti, l'unica chiave sarà *nomePlugin.DiESeL.Powered*;
- nel caso in cui il nodo cerchi se stesso le chiavi saranno *nomePlugin.DiESeL.Powered IP/porta1/porta2* e *nomeServer*;[1]

Di seguito è riportata la porzione di codice che effettua la richiesta alla *Distributed Hash Table*:

```
[...]
IFeatureValue[] searchValues = new FeatureValue[1];
searchValues[0] = new FeatureValue("time", 30000);
IRequest req = new Request(reqestForAPI, new
    ConstructorParameters(searchValues, DHTAPI.Primitive.
    PPFIND_PRIMITIVE, keys));
IReply reply = null;
try {
    reply = myGrandFather.askTheCore(req);
}
[...]
```

Keys è la lista delle chiavi che verranno ricercate.

p2pCommunication Questo metodo contatta tutti i nodi appartenenti alla lista ottenuta in seguito al metodo *interrogationToDHT*, ritornando le coppie *Nodo / Note* che soddisfano i requisiti della ricerca.

returnListNodeResults Dalle coppie *Nodo / Note* del metodo precedente, vengono estratti gli indirizzi IP dai nodi, e dalle Note, le porte e le Feature, ritor-

[1] Il nodo ricerca se stesso per controllare di essere ancora pubblicato nella *DHT*; in caso contrario provvede ad eseguire lo storage dei dati.

nando una lista di *DiESeLNodeExtended* che soddisfano i requisiti richiesti, ovvero le chiavi cercate.

returnListServerName Il metodo restituisce la lista dei nomi dei server esistenti; tali nomi vengono estratti dalle coppie *Nodo / Note* del metodo precedente.

La ricerca da parte del comandante di nodi da aggiungere sarà fatta quindi mediante l'utilizzo in sequenza dei metodi *interrogationToDHT*, *p2pCommunication* e *returnListNodeResults*; successivamente i nodi trovati saranno contattati direttamente, unendoli al server.

La ricerca dei nomi dei server esistenti invece utilizza *interrogationToDHT*, *p2pCommunication* e *returnListServerName*.

6.2 Features

L'aggiunta delle *Features* ha comportato la realizzazione di diverse classi e la riprogettazione di altre. La prima implementata è stata quella che andrà a contenere la singola caratteristica, la quale verrà utilizzata per contenere tutte le *Features* in una ArrayList di Java.

6.2.1 Features: la struttura dati e l'IDiESeLNodeExtended

La memorizzazzione delle caratteristiche del server e dei nodi avviene mediante il loro inserimento in una ArrayList di oggetti denominati *Feature*. Questa classe non è altro che una coppia *Stringa / Valore*, in cui saranno inseriti rispettivamente il nome della caratteristica e il suo valore intero o in virgola mobile. Nel caso del server, le sue caratteristiche verranno mantenute semplicemente in una ArrayList di

Features, mentre nel caso delle caratteristiche dei nodi, si è scelto di incapsulare tale ArrayList in una classe che estende quella in cui sono memorizzati gli identificativi dei nodi della libreria, denominata *DiESeLNodeExtended*. Ricordiamo che gli *DiESeLNode* contengono l'indirizzo IP e le due porte necessarie alla comunicazione dei soket. In figura 6.1 si può vedere la rappresentazione della nuova classe.

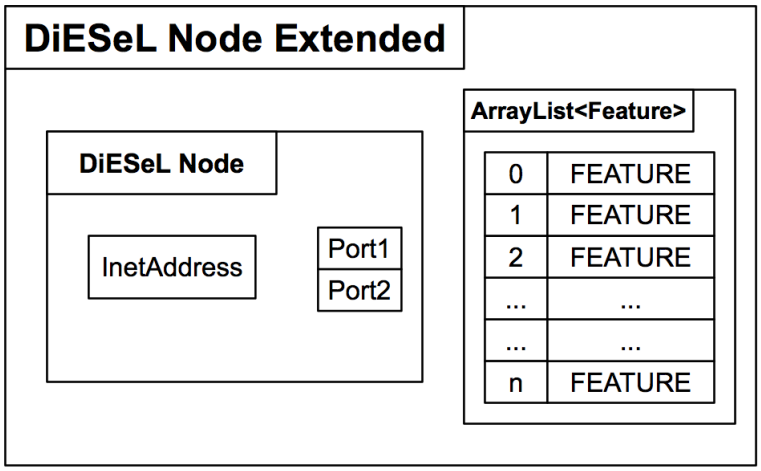

Figura 6.1: Rappresentazione della classe DiESeLNodeExtended, che contiene un DiESeLNode e una lista di Feature.

DiESeLNodeExtended implementa l'omonima interfaccia, la quale estende l'interfaccia *IDiESeLNode*. Questo permette di non andare a modificare gran parte del codice: la classe *Neighborhood*, per esempio, mantiene i *DiESeLNode* nei riferimenti interfaccia a *IDiESeLNode*, e quindi senza refactoring è possibile mantenere anche gli *DiESeLNodeExtended*.

6.2.2 Comunicazione con il ServerLayer

Per mantenere la massima indipendenza dal Plugin che utilizza la libreria, e la massima generalità, si è lasciato il compito di richiedere le *Features* all'utente, al *ServerLayer*. E' quest'ultimo che dovrà controllare la bontà dei dati fornitagli dagli

utenti, per evitare che vi siano dichiarazioni fallaci che porterebbero a situazioni potenzialmente pericolose. Il *ServerLayer* dovrà valutare sia le specifiche richieste del server che la disponibilità dichiarata di ciascun nodo. Si noti che questi controlli non dovranno essere fatti nel caso in cui i nodi vengano attivati via codice.

Come già detto le *Features* non sono note a priori, quindi la comunicazione con il *ServerLayer* avverrà mediante un file *.xml*, particolarmente adatto per la sua natura estensibile. I vari Plugin che utilizzeranno *DiESeL* dovranno semplicemente compilare un file con la struttura uguale a quella che si può trovare in figura 6.2, il quale verrà letto dalla libreria.

```
<?xml version="1.0"?>
<featureDefinition>
    <feature>
        <name>NameFeature1</name>
        <value>1000000</value>
    </feature>
    <feature>
        <name>NameFeature2</name>
        <value>3000000</value>
    </feature>
    <feature>
        <name>NameFeature3</name>
        <value>6000000</value>
    </feature>
</featureDefinition>
```

Figura 6.2: File *.xml* in cui vengono specificate le *Features*. Si noti che il nome delle *Features* ed il loro valore sono gli unici campi che è possibile modificare.

Per adempiere a questo compito, è stata scritta una classe apposita: *XmlFeatureParser*. Questa va ad implementare l'omonima interfaccia, la quale estende l'interfaccia più generale *IFeatureParser*. Si è pensato di procedere in questa maniera in modo da dare la possibile agli sviluppatori futuri di utilizzare file di estensione differente.

Una volta lette, le Feature dovranno essere pubblicate nella *DHT*, insieme alle note riguardanti il nodo; questo passo risulta necessario per poter applicare la politica di aggregazione che verrà descritta nel sottoparagrafo successivo (si veda 6.2.3). Se non si procedesse in questa maniera, il nodo attivo non potrebbe immediatamente distinguere i nodi, ma dovrebbe andare a contattarli uno ad uno per decidere, in base all'analisi delle *Features*, quale sia il migliore in quel dato momento. Questo modo di procedere comporterebbe un rallentamento della procedure, oltre ad un traffico aggiuntivo e conseguente spreco di banda.

6.2.3 Politica di aggregazione

Come spiegato nel precedente sottoparagrafo, le informazioni necessarie alla scelta del nodo da aggiungere si trovano sulla *DHT*, e i metodi per la richiesta sono gli stessi analizzati nel paragrafo 6.1. Attraverso il metodo *checkForOtherNode()* della classe *Distributore*, saranno ricercati tutti i nodi aggregabili al server e se ve ne saranno, e se il server non sarà completo, verrà invocato il metodo *newNodeRequest()*.

NewNodeRequest() ha due compiti:

- scegliere il nodo da unire;
- contattare il prescelto.

Data la lista dei nodi da aggiungere, e le rispettive *Features*, la politica di aggregazione si basa sul concetto di multidimensionalità: ogni caratteristica si può pensare come una dimensione di uno spazio, definito dalla lista delle caratteristiche specificate dal creatore del server. Per esempio, nel caso in cui si vorrà realizzare

un nuovo server che avrà quattro *Features*, al momento della scelta sull'elemento da aggregare, ci si troverà in uno spazio quadridimensionale. Ogni lista di *Features*, andrà quindi a definire un punto in tale spazio; si noti che sia le specifiche richieste per il server, sia le *Features* di ciascun nodo, sono identificate da un punto. Per comodità, nella classe principale della libreria, ovvero in *Distributore*, sarà mantenuta la lista (e quindi il punto) delle caratteristiche mancanti al server, aggiornata ogni qual volta viene aggiunto o eliminato un nodo.

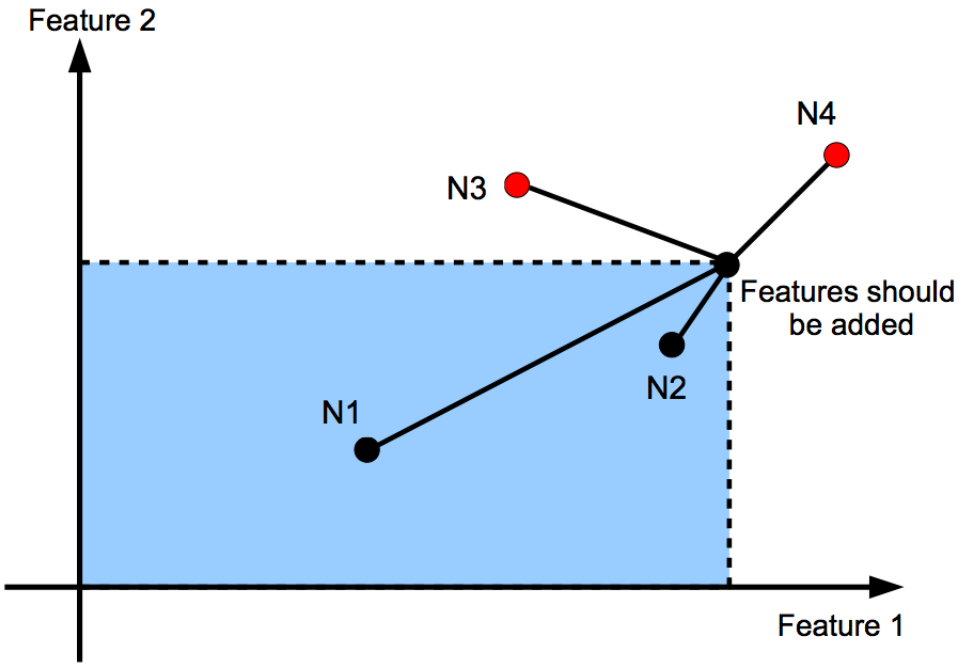

Figura 6.3: Esempio della politica di aggregazione implementata: in questo caso sono presenti solo due caratteristiche e la scelta del nodo da aggiungere sarà fatta in base alla distanza dal punto contrassegnato con il nome *"Features should be added"*. Come si vede dall'immagine a questo passo dell'algoritmo verrà scelto il nodo N2. Si noti che al passo successivo non verrà unito il nodo N4 (più vicino degli altri alla richiesta di *Features* del srever), in quanto il punto *"Features should be added"* si sposterà a causa dell'unione di N2. Quindi in questo caso, sarà aggiunto N1.

La scelta di chi sarà aggiunto verrà quindi effettuata in base alla minor distanza

tra il punto determinato dalla quantità mancante di *Features* al server, e il punto definito dalla disponibilità di ciascun nodo della lista. Questa politica risulta essere buona, in quanto si avrà la garanzia che sarà scelto il nodo che massimizzerà i benefici e che ridurrà gli sprechi. In figura 6.3 si può vedere un esempio pratico, in cui sono presenti solo due caratteristiche. Con *"Features should be added"* è identificato il punto rappresentante il quantitativo di *Features* che devono essere sommate a quelle del server per completarlo. Se un nodo sta nella zona colorata (N1, N2) significa che se venisse scelto, non supererebbe la richiesta di *Features* e quindi non ci sarebbero sprechi. Invece i nodi posti al di fuori della zona colorata (N3, N4), se scelti, aggiungerebbero un quantitativo superiore a quello richiesto. L'algoritmo comunque garantisce che, anche nel caso in cui si aggiungano più *Features* di quelle richieste, sarà scelto l'elemento che ne sprecherà meno, ovvero quello con minor distanza dal punto *"Features should be added"*. Nel caso in cui, per scelta del creatore del server, non sono specificate le *Features*, viene aggregato un nodo a caso.

Il metodo che determina quale elemento verrà unito è *findBestNodesToAdd* della classe *Distributore*.

Una volta deciso chi sarà aggiunto, si procederà a contattarlo, inviandogli, come già spiegato nel paragrafo 3.5, i nodi a cui unirsi e il tempo del server.

6.2.4 Come mantenere le Features

Per rendere attuabile l'algoritmo di aggregazione descritto al paragrafo precedente, è fondamentale che il nodo attivo mantenga le Feature di tutti i partecipanti al server, in modo da conoscere in ogni momento, anche dopo eventuali cadute,

lo stato del server. Si è quindi deciso di mantenerle nella Neighborhood, classe che contiene le informazioni di tutti gli elementi del server, e per ciascuno di essi, tutti i vicini. La Neghborhood utilizza come struttura dati per memorizzare i nodi, una Hash Table composta da coppie di oggetti *Chiave-Lista* dove la chiave è un elemento del server, e la lista è l'insieme di tutti i nodi a cui la chiave è connessa.

Al fine di non utilizzare troppa memoria, si è deciso di mantenere le *Features* non in ciascuna occorrenza del nodo stesso nella Neighborhood, ma solo dove compare come *Chiave*, e quindi un unica volta. Questo passo non ha modificato la struttura della tabella, come si è già spiegato nel paragrafo 6.2.1. Infatti mantenendo i nodi nei riferimenti interfaccia a *IDiESeLNode*, senza refactoring è possibile mantenere anche i *DiESeLNodeExtended*. Nella classe sono stati aggiunti solo alcuni metodi per la gestione delle *Features* tra cui:

getDroppedNeighborhood Restituisce la Neighborhood senza le *Features* dei nodi. Avrà quindi come chiavi dei *DiESeLNode*. permettendo di inviare la tabella ai nuovi entrati in maniera più rapida.[2]

getNodeWithFeature Ritorna la lista dei nodi della tabella con le *Features*.

getNodeWithOutFeature Ritorna la lista dei nodi della tabella che non hanno le *Features* ancora settate.

FeaturesAreSet Ritorna il valore booleano *Ture* se le *Features* di uno specifico nodo sono memorizzate al completo nella Neighborhood, *False* altrimenti.

Al comandante basterà quindi invocare il metodo $getNodeWithFeature$ per conoscere lo stato del server.

[2]Non avendo posto dei limiti alle *Features*, potrebbero essere anche molto pesanti, comportando un traffico aggiuntivo non sostenibile.

6.2.5 Algoritmo di distribuzione

Come spiegato nel precedente capitolo, è necessario che anche i nodi passivi vengano a conoscenza delle *Features* di tutti gli altri componenti del server, in modo da poter gestire in maniera rapida e senza traffico aggiuntivo, la disconnessione accidentale o volontaria del comandante. Per farlo si utilizza una strategia che si ispira agli algoritmi epidemici. In particolare si andranno ad inserire le *Features* nel piggyback dei ping utilizzati dall'algoritmo di Keep Alive, aumentandone di pochi KByte la taglia.

Per realizzare quest'algoritmo sono stata implementate due classi che ora verranno analizzate.

RandomizedQueue

La prima classe fondamentale per la riuscita dell'algoritmo è la RandomizedQueue (figura 6.4), una coda in cui sarà inserita la lista dei *DiESeLNode* di cui sono note le *Features*. Inizialmente l'unico nodo presente sarà il "proprietario" dell'istanza di *DiESeL*, poiché le uniche *Features* conosciute saranno le sue stesse. Nel caso sia attivo (A), quando aggregherà un altro nodo, inserirà le sue caratteristiche nella RandomizedQueue. Ogni qualvolta A andrà ad eseguire un ping del CKA, estrarrà dalla coda il primo elemento, recupererà le sue *Features* mediante un interrogazione alla Neighborhood, e le inserirà nel piggyback. Il nodo che riceverà questo ping (B), estrarrà dal piggyback le *Features* ed andrà ad inserirle nella sua Neighborhood, aumentando così la sua conoscenza della rete. Nel momento in cui vengono scoperte nuove caratteristiche di un nodo, questo verrà inserito nella RandomizedQueue in una posizione casuale, in modo che nei prossimi ping, B potrà

inviare ai suoi vicini le *Features* arrivategli da A. Comportandosi tutti i nodi della rete in questa maniera si avrà la garanzia che in un certo numero di passi ognuno avrà una conoscenza completa delle caratteristiche dei parteciapanti al server.[5] Il numero di passi dipende dalla taglia del piggyback che si sceglie, e dalla taglia delle *Features*: se in un pigyback si riescono ad inserire più caratteristiche di più nodi, la convergenza sarà molto più rapida rispetto al caso in cui si inserisca solo una parte delle *Features* di un nodo. La taglia dei ping utilizzati nell'algoritmo di Keep Alive è di circa 800 byte. Si è scelto di andare a caricare il piggyback di non più del 25%, inserendovi quindi non più di 200 byte di *Features*. Così facendo il ping non è troppo appesantito, e si ha un risparmio notevole in termini di spreco di banda che sarebbe causato dall'intestazione del ping. Il tempo di convergenza dipenderà anche dal numero di partecipanti al server.

Nella figura 6.4 si può vedere la coda dei nodi, le cui Feature verranno inviate nei ping. Nel caso in cui in un messaggio non venga inviata la lista completa di *Features*, ma solo una sua parte, l'indice dell'ultima caratteristica inviata sarà memorizzato in *LastFeatIndex*. Quindi ad ogni estrazione dalla coda si dovrà controllare quest'indice, e se uguale a -1 significa che si dovrà inviare una nuova lista di *Features*, altrimenti si dovrà inviare la lista di caratteristiche del nodo in testa alla coda a partire da da quella in posizione *LastFeatIndex*.

I metodi della classe sono i seguenti:

get Restituisce il primo elemento della lista.

pop Come get, con la differenza che l'elemento viene eliminato dalla lista.

getIndex Ritorna *LastFeatIndex*.

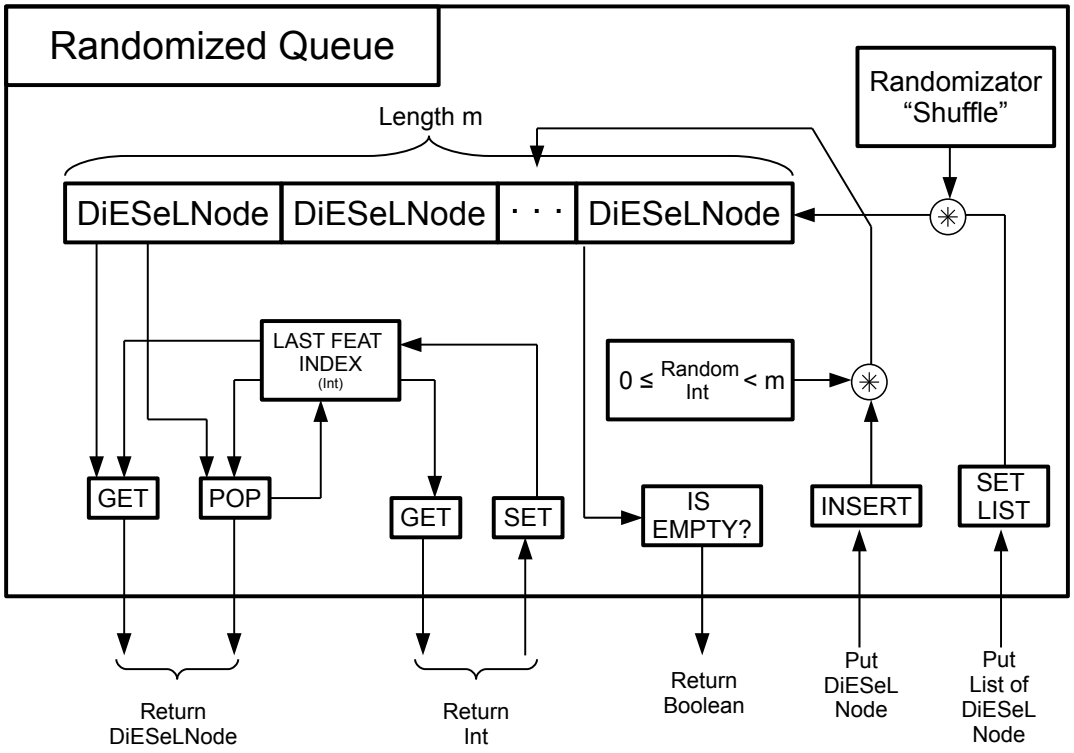

Figura 6.4: Rappresentazione della classe RandomizedQueue.

setIndex Setta l'*LastFeatIndex* al valore voluto.

insert Può venire utilizzato per inserire nella lista o un unico nodo (è il caso in cui il nodo viene a conoscenza delle *Features* di un elemento) o una lista di nodi (è il caso in cui la lista si svuota, e quindi si vanno a reinserire tutti gli elementi di cui si conoscono le *Features*).

isEmpty Ritorna *True* se la coda è vuota, *False* altrimenti.

Sarà necessario mantenere una RandomizedQueue per ogni nodo a cui si è connessi, in modo da gestire in maniera differente le *Features* da inviare, riducendo

il tempo di propagazione e gli sprechi di banda, a discapito di un utilizzo maggiore di memoria.

Struttura generale e PairNodeQueue

Come detto, per ogni elemento a cui si è connessi nella sottorete di *DiESeL*, ci sarà una RandomizedQueue, e per gestirle si è implementata una classe composta da coppie *Nodo di destinazione / RandomizedQueue*. Vista la stretta relazione con la classe CKASender, è stata inserita in quest'ultima come classe privata, con il nome di PairNodeQueue. In figura 6.5 si può vedere la procedura per effettuare un ping: viene estratto il primo ping dalla lista e se ne determina il destinatario; questo sarà utilizzato per accedere tramite la PairNodeQueue alla RandomizedQueue corrispondente. Tramite il metodo POP o GET, descritti al sottoparagrafo precedente, si andrà a inserire nel piggyback le Feature. Nel caso in cui vi sia ancora spazio nel ping, si accederà nuovametne alla RandomizedQueue, inseredo altre Feauture.

La taglia del piggyback andrà ad incidere sulla velocità di convergenza dell'algoritmo; per non appesantire troppo i ping, che come detto sono di circa 800 byte ciascuno, si è deciso di dimensionare il carico massimo da aggiungervi a non più di 200 byte.

Ogni qual volta il nodo viene a conoscenza delle Feature di un altro partecipante alla rete, le salverà nella Neighborhood, e successivamente andrà ad inserire in ciascuna RandomizedQueue, in posizione casuale, l'identificativo del nodo. Così facendo si avrà la certezza che nei ping seguenti, diffonderà la sua conoscenza anche ai vicini non in comune.

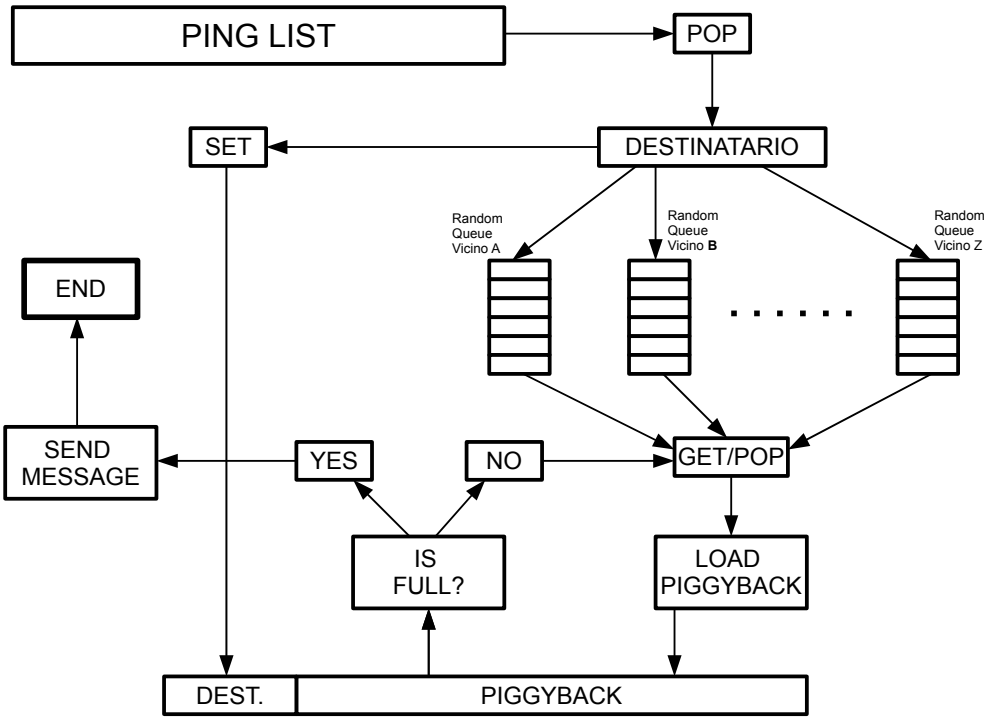

Figura 6.5: Algoritmo di composizione di un ping che inserirà nel piggyback le Feature da comunicare ai vicini.

Capitolo 7

Conclusione e sviluppi futuri

7.1 Conclusioni

In questa tesi sono state descritte le attività che hanno portato *DiESeL*, una libreria che permette di distribuire un server sulla rete *PariPari*, a riuscire a far cooperare un numero arbitrario di nodi. Inizialmente il numero massimo di elementi era tre, oltre il quale il server si disuniva, portando alla sua caduta. Inoltre, ora i nodi del server riescono a gestire la disconnessione di un elemento, cosa molto frequente data la natura della rete, dopo una breve fase di riassestamento. Successivamente si è passati all'aggiunta di due elementi che sono andati ad aumentare la potenza di *DiESeL*: il Multiserver e le *Features*.

Il Multiserver da la possibilità ai Plugin che utilizzeranno la libreria di andare a distribuire più server, a differenza di prima, dove era possibile crearne uno solo a cui tutti i nodi del Plugin si connettevano. Le *Features* invece, sono delle caratteristiche che qualificano sia i server che i nodi che vi partecipano.

La loro implementazione ha reso possibile inserire una politica di aggregazione (in precedenza non si potevano distinguere i nodi e quindi si univano tutti), in cui si connetteranno prima i nodi più adatti, e solo in un secondo momento, e se le specifiche del server lo richiedono, gli altri meno meritevoli. La dichiarazione delle caratteristiche che dovrà avere il server permette inoltre di andare a ridurre gli sprechi, in quanto una volta raggiunta la quota di *Features* richiesta, non verranno più aggiunti elementi. Questi attenderanno finché o uno dei nodi facenti parte del server si disconnette, oppure sino a quando un server, definito dagli sviluppatori del Plugin con una join policy particolare, non li contatterà.

7.2 Sviluppi futuri

Nei capitoli precedenti sono emersi alcuni punti che necessitano di uno sviluppo in futuro. Il primo è la ricerca di un algoritmo efficiente per il broadcasting sulla sottorete del server distribuito, sfruttando la topologia nota della rete stessa al fine di minimizzare il traffico inoltrato. Tale compito però spetterebbe al gruppo di Connectivity. Il secondo riguarda l'interazione con il DNS di *PariPari*, in quanto al momento le disconnessioni esterne al server non sono gestite. Il terzo riguarda l'indipendenza di *DiESeL*, promuovendo al libreria a un Plugin a tutti gli effetti, facilitandone l'utilizzo da parte degli altri Plugin.

Oltre a questo sarà necessario un lavoro continuo di manutenzione e testing, al fine di aggiornare la libreria di pari passo con l'evoluzione di PariPari. In particolare sarebbe utile testare *DiESeL* su larga scala, in quanto le prove effettuate sono

state fatte con un numero massimo di nodi pari a dieci. Sarà necessario che sia mantenuta una documentazione sempre aggiornata (utilizzando JavaDoc e la Wiki di PariPari) e un costante aggiornamento, sia tra i membri interni al team di sviluppo specifico, sia con lo staff generale di PariPari attraverso le periodiche riunioni tra i capigruppo.

Bibliografia

[1] P. Bertasi (2005), *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale*, Tesi Università degli studi di Padova.

[2] A. Marcassa (2008), *PariPari: IRC Server Distribuito*, Tesi Università degli studi di Padova.

[3] M. Jelasty, A. Montresor, O. Babaoglu (2005), *Gossip-Based Aggregation in Large Dynamic Network*, ACM Transactions on Computer System, Vol. 23, No. 3, August 2005.

[4] A. Montresor, A. Ghodsi (2009), *Towards Robust Peer Counting*, IEEE P2P Conference, `http://napa-wine.eu`.

[5] M. Jelasty, A. Montresor, O. Babaoglu (2009), *T-MAN: Gossip-based fast overlay topology construction*, Computer Networks 53, `http://www.sciencedirect.com/science/journal/13891286`.

[6] P. Maymounkov, D. Mazières (2002), *A Peer-To-Peer Information System Based on the XOR Metric*, Lecture Notes in Computer Science, Volume 2429/2002.

[7] M. Ripeanu, I. Foster, A. Iamnitchi (2002), *Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design*, IEEE Internet Computing 6.

[8] I. Dedinski, A. Hofmann, B. Sick (2007), *Cooperative Keep-Alives: An Efficient Outage Detection Algorithm for P2P Overlay Networks*, Seventh IEEE International Conference on Peer-to-Peer Computing.

Elenco delle figure